AD-A198 884

DTIC FILE COPY

(2)

AVF Control Number: NBS87VCDC535

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 870122S1.09018
Control Data Corporation
CYBER 180 Ada Compiler, Version 1.0
HOST and TARGET COMPUTER:
CYBER 180-830

Completion of On-Site Testing:
22 January 1988

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Ada Compiler Validation Summary Report: Control Data Corporation,CYBER 180 Ada Compiler, Version 1.0, CYBER 180-830 (Host and Target).		5. TYPE OF REPORT & PERIOD COVERED 22 Jan 1988 to 22 Jan 1989 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		12. REPORT DATE 22 January 1988 13. NUMBER OF PAGES 64 p.
14. MONITORING AGENCY NAME & ADDRESS *(If different from Controlling Office)* National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		15. SECURITY CLASS *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20. If different from Report)*

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS *(Continue on reverse side if necessary and identify by block number)*

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

CYBER 180 Ada Compiler, Version 1.0, Control Data Corporation, National Bureau of Standards, U.S.A. CYBER 180-830 under NOX/VE, Level 688 (Host and Target),ACVC 1.9.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Ada Compiler Validation Summary Report:

Compiler Name: CYBER 180 Ada Compiler, Version 1.0

Certificate Number: 880122S1.09018

Host:	Target:
CYBER 180-830 under NOS/VE, Level 688	CYBER 180-830 under NOS/VE, Level 688

Testing Completed 22 January 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the CYBER 180 Ada Compiler, Version 1.0, using Version 1.9 of the Ada Compiler Validation Capability (ACVC). The CYBER 180 Ada Compiler is hosted on a CYBER 180-830 operating under NOS/VE, Level 688. Programs processed by this compiler were executed on a CYBER 180-830 operating under NOS/VE, Level 688.

On-site testing was performed 18 January 1988 through 22 January 1988 at Sunnyvale, California, under the direction of the Software Standards Validation Group, Institute for Computer Sciences and Technology. National Bureau of Standards (AVF), according to Ada Validation Organization (AVO) policies and procedures. At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 118 were determined to be inapplicable to this implementation. Results for processed Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. The remaining 2979 tests were passed. The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	187	559	640	242	166	98	139	326	135	36	232	3	216	2979
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	14	35	6	0	0	4	1	2	0	2	0	37	118
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

> To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
>
> To attempt to identify any unsupported language constructs required by the Ada Standard
>
> To determine that the implementation-dependent behavior is allowed by the Ada Standard

On-site testing was conducted from 18 January 1988 through 22 January 1988 at Sunnyvale, California.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Software Standards Validation Group
> Institute for Computer Sciences and Technology
> National Bureau of Standards
> Building 225, Room A266
> Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should

be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. <u>Reference Manual for the Ada Programming Language</u>, ANSI/MIL-STD-1815A, February 12989.

2. <u>Ada Compiler Validation Procedures and Guidelines</u>. Ada Joint Program Office, 1 January 1987.

3. <u>Ada Compiler Validation Capability Implementers' Guide</u>. SofTech, Inc., December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 12989.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established procedures.
AVO	The Ada Validation Organization. In the context of this, report, the AVO is responsible for establishing procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective had been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: CYBER 180 Ada Compiler, Version 1.0

ACVC Version: 1.9

Certificate Number: 880122S1.09018

Host Computer:

Machine:	CYBER 180-830
Operating System:	NOS/VE Level 688
Memory Size:	24Mbytes RAM

Target Computer:

Machine:	CYBER 180-830
Operating System:	NOS/VE Level 688
Memory Size:	24Mbytes RAM

Additional Configuration Information:

2.7 Gbytes disk drives
terminals connected using CDCNET
5 magnetic tape drives
2000 l/m printer

Communications Network: none

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types LONG_FLOAT in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently all default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

CONSTRAINT_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array subtype is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than

INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

The Ada Standard does not require an implementation to support

representation clauses. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are not supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

There are strings which are illegal external file names for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102C and CE2102H.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can not be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

Only one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE2110B, CE2111D, CE3111A..E (5 tests), CE3114B, and CE3115A.)

Only one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests) and CE2111D.)

Only one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107E, CE2107G..I (3 tests) and CE2111H.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See test CE2107F.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are not given names. Temporary direct files are not given names. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can compiled in separate compilations so long as no instantiations of those units precede the bodies. This compiler requires that a generic

unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected. (See test CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations so long as no instantiations of those units precede the bodies. This compiler requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected. (See tests CA2009C, BC3204C.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 118 were determined to be inapplicable to this implementation. Modifications to the code, processing, or grading for 67 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	105	1043	1754	17	14	46	2979
Failed	0	0	0	0	0	0	0
Inapplicable	5	8	101	0	4	0	118
Withdrawn	3	2	19	0	1	0	25
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	187	559	640	242	166	98	139	326	135	36	232	3	216	2979
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	17	14	35	6	0	0	4	1	2	0	2	0	37	118
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 25 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35A03E	C35A03R	C37213H	C37213J	C37215C	C37215E
C37215G	C37215H	C38102C	C41402A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A	CE2401H
CE3208A					

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 118 test were inapplicable for the reasons indicated:

C24113I..X (16 tests) contain contain lines whose length is greater than this implementation's maximum line length of 132 characters.

C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1). These clauses are not supported by this compiler.

C35702A uses SHORT_FLOAT which is not supported by this implementation.

A39005B and C87B62A use length clauses with SIZE specifications for derived integer types or for enumeration types which are not supported by this compiler.

A39005E and C87B62C use length clauses with SMALL specifications which are not supported by this implementation.

A39005G uses a record representation clause which is not supported by this compiler.

The following (15) tests use SHORT_INTEGER, which is not supported by this compiler.

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001CR

The following (13) tests use LONG_INTEGER, which is not supported by this compiler.

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

C45231D requires an integer type other than INTEGER.

C4A013B uses a static value that is within the range of the most accurate floating point base type, and MACHINE_OVERFLOWS is false for this type. The test executes and reports NOT_APPLICABLE.

B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation there are no values between DURATION'FIRST and DURATION'BASE'FIRST.

CA2009C compiles the bodies of generic units separately and following a compilation that contains instantiations of those units. This compiler requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

CA2009F compiles the bodies of generic units separately and following a compilation that contains instantiations of those units. This compiler

requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

BC3204C compiles the bodies of generic units separately and following a compilation that contains instantiations of those units. This compiler requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

BC3205D compiles the bodies of generic units separately and following a compilation that contains instantiations of those units. This compiler requires that a generic unit's body be compiled prior to instantiation, and so the unit containing the instantiations is rejected.

AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

CE2102E this implementation supports mode OUT_FILE for SEQUENTIAL_IO.

CE2102F this implementation supports mode INOUT_FILE for DIRECT_IO.

CE2102G this implementation supports RESET for SEQUENTIAL_IO.

CE2102J this implementation supports mode OUT_FILE for DIRECT_IO.

CE2102K this implementation supports RESET for DIRECT_IO.

CE2105A..B (2 tests), CE2111H, CE2407A, CE3104A, CE3109A this implementation does not allow the creation of a file when FILE_MODE is set to IN_FILE.

CE2107A..C (3 tests) this implementation does not allow more than one internal sequential file to be associated with the same external file.

CE2107D..E (2 tests), CE2107H..I (2 tests), CE2108A, CE2108C, CE3112A this implementation does not allow temporary files to have a name.

CE2107G this implementation does not allow more than one internal direct file with mode OUT_FILE or mode INOUT_FILE to be associated with the same external file.

CE2110B, CE2111D, CE3111A..E (5 tests), CE3114B, CE3115A this implementation does not allow more than one internal text file to be associated with the same external file.

The following 19 tests require a floating-point accuracy that exceeds the maximum of 28 digits supported by this implementation:

C24113Y	(01 test)	C35705Y	(01 test)
C35706Y	(01 test)	C35707Y	(01 test)
C35708Y	(01 test)	C35802Y..Z	(02 tests)
C45241Y	(01 test)	C45321Y	(01 test)
C45421Y	(01 test)	C45521Y..Z	(02 tests)
C45524Y..Z	(02 tests)	C45621Y..Z	(02 tests)
C45641Y	(01 test)	C46012Y..Z	(02 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made with the approval of the AVO, and are made in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 67 Class B tests (71 test files).

The following Class B test files were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B26001A	B26002A	B26005A	B28001D	B28003A
B29001A	B2A003A	B2A003B	B2A003C	B33301A	B35101A
B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B51001A	B53009A	B54A01C	B54A01J	B54A01K
B55A01A	B61001C	B61001D	B61001F	B61001M	B61001R
B61001W	B66001C	B67001A	B67001C	B67001D	B91001A
B91002A	B91002B	B91002C	B91002D	B91002E	B91002F
B91002G	B91002H	B91002I	B91002J	B91002K	B91002L
B95030A	B95061A	B95061F	B95061G	B95077A	B97101A
B97101E	B97102A	B97103E	B97104G	BA1101BOM	BA1101B1
BA1101B2	BA1101B3	BA1101B4	BC1109A	BC1109C	BC1109D
BC1202A	BC1202B	BC1202G	BC2001D	BC2001E	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the CYBER 180 Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the CYBER 180 Ada Compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a CYBER 180-830 operating under NOS/VE, Level 688; the host and targe computers were the same.

A magnetic tape containing all tests except for withdrawn tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the CYBER 180-830, and all executable tests were run on the CYBER 180-830.

The compiler was tested using command scripts provided by Control Data Corporation and reviewed by the validation team. The compiler was tested using all default (option/switch) settings except for the following:

Option/Switch	Effect
INPUT	Name of the source text file
PROBRAM LIBRARY	Name of the program library
LIST	Name of the source listing file
ERROR	Name of the error file

Tests were compiled, linked, and executed using a single host/target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF.

3.7.3 Test Site

The validation team arrived at Sunnyvale, California on 18 January 1988, and departed after testing was completed on 22 January 1988.

APPENDIX A

CONFORMANCE STATEMENT

APPENDIX A

CONFORMANCE STATEMENT

Compiler Implementor:

Control Data Corporation
215 Moffett Park Dr.
Sunnyvale, CA 95089

Ada Validation Facility:

Software Standards Validation Group
Institute for Computer Science and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Base Configuration

Base Compiler Name: CYBER 180 ADA Compiler, Version: 1.0
Host Architecture ISA:
CYBER 180 - 830 OS&VER #: NOS/VE LEVEL 688

Implementor's Declaration

I, the undersigned, representing Control Data Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Control Data Corporation is the owner of record of the ///ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler listed in this declaration shall be made only in the owner's name:

Control Data Corporation Date: December 23, 1987

Ada* is a registered trademark of the United States Government (Ada Joint Program Office).

Richard J. Clifton, Manager

Owner's Declaration

I, the undersigned, representing Control Data Corporation, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Date: December 23, 1987

Control Data Corporation

Richard J. Clifton, Manager

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the CYBER 180 Ada Compiler, Version 1.0, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A).. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is


     type INTEGER is range -9_223_372_036_854_775_808 ..
                            9_223_372_036_854_775_807;

     type FLOAT is digits 13 range
                        -16#7.FFFF_FFFF_FFF8#E1023..
                         16#7.FFFF_FFFF_FFF8#E1023;

     type LONG_FLOAT is digits 28
                        -16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023..
                         16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023;

     type DURATION is delta 0.001 range
                        -6_279_897_600.0 .. 6_279_897_600.0;

end STANDARD;
```

Implementation-Dependent Characteristics F

This appendix summarizes the implementation-dependent characteristics (*IDCs*) of NOS/VE Ada by listing the following:

- NOS/VE Ada pragmas
- NOS/VE Ada attributes
- Specification of the package SYSTEM
- Restrictions on representation clauses
- Conventions for implementation-generated names denoting implementation-dependent components
- Interpretation of expressions appearing in address clauses
- Restrictions on unchecked type conversions
- Implementation-dependent characteristics of input-output packages
- Other implementation characteristics

FIRST DRAFT INTERNAL USE ONLY

F.1 NOS/VE Ada Pragmas

NOS/VE Ada supports all the pragmas required by the ANSI standard for Ada. It does not provide any implementation defined pragmas.

NOS/VE Ada does not support the following pragmas

- CONTROLLED
- MEMORY_SIZE
- OPTIMIZE
- PRIORITY
- STORAGE_UNIT
- SYSTEM_NAME

NOS/VE Ada supports the following pragmas as described in the ANSI Reference Manual for Ada, except as shown below:

- INLINE

  This pragma causes inline expansion of a subroutine except as described in Annex B of this manual. See 10.6 to see how to use INLINE to create faster object code.

- INTERFACE

  This pragma is supported for FORTRAN, CYBIL, and the NOS/VE MATH_LIBRARY, as discussed in 13.9.1IDC, 13.9.2IDC, and 13.9.3IDC, respectively.

- PACK

  Objects of the given type are packed into the nearest 2**n bits.

- SHARED

  This pragma is not allowed for the following types of variables:

  - Type LONG_FLOAT
  - Variables of subtype of type LONG_FLOAT
  - Variables of a type derived from type LONG_FLOAT
  - Variables of a subtype derived from type LONG_FLOAT

- SUPPRESS

  NOS/VE supports this pragma, but it is not possible to restrict the check suppression to a specific object or type.

F.2 NOS/VE Ada Attributes

NOS/VE Ada supports all the attributes required by the ANSI Ada language definition. It does not provide any implementation defined attributes. The NOS/VE implementation of the P'ADDRESS attribute returns for the prefix P, the 48-bit Program Virtual Address (PVA) right justified within a 64-bit INTEGER.

F.3 Specification of the Package SYSTEM

```
package SYSTEM is

   type ADDRESS is access INTEGER;

   type NAME is (CYBER180);

   SYSTEM_NAME : constant NAME := CYBER180;
   STORAGE_UNIT : constant := 64;
   MIN_INT : constant := -9_223_372_036_854_775_808;
   --(-2**63)

   MAX_INT : constant := 9_223_372_036_854_775_807;
   --(2**63)-1

   MAX_DIGITS : constant := 28;
   MAX_MANTISSA : constant := 63;
   FINE_DELTA : constant := 2#1.0#E-63; --2**(-63)
   TICK : constant := 0.001;

   subtype PRIORITY is INTEGER range 0..0;

end SYSTEM;
```

F.4 Restrictions on Representation Clauses

NOS/VE Ada implements representation clauses as required by the ANSI standard for Ada. It does not allow representation clauses for a derived type.

NOS/VE Ada supports all of the type representation clauses:

- Length clauses
- Enumeration representation clauses
- Record representation clauses

NOS/VE Ada does not support address clauses or interrupts.

F.4.1 Length Clauses

NOS/VE Ada supports the attributes in the length clauses as follows:

- T'SIZE

  Not supported

- T'STORAGE_SIZE (collection size)

  Supported

- T'STORAGE_SIZE (task activation size)

  Supported

FIRST DRAFT INTERNAL USE ONLY

- T'SMALL

  Not supported. The compiler always chooses for SMALL the largest power of 2 not greater than the delta in the fixed accuracy definition of the first named subtype of a fixed point type.

  For example, NOS/VE Ada uses the declaration:

  type ADA_FIXED **is delta** 0.05 **range** 1.00 .. 3.00..;

  to set the ADA_FIXED'SMALL attribute to 0.03125(2^{-5}, the largest power of 2 not greater than **delta**, 0.05.

F.4.2 Record Representation Clauses

NOS/VE Ada inplements record representation clauses as required by the ANSI language definition. It does not support alignment clauses in record representation clauses.

The component clause of a record representation clause gives the storage place of a component of the record, by providing the following pieces of data:

- The name gives the name of the record component.
- The simple expression following the reserved word AT gives the address in storage units, relative to the beginning of the record, of the storage unit where the component starts.
- The range in the component clause gives the bit positions, relative to that starting storage unit, occupied by the record component.

NOS/VE Ada supports the range for only those record components of discrete type: integer or enumeration. Also the range must specify 16 bits or fewer.

Furthermore, if the range specifies 8 or more bits, then:

- The starting storage unit address must be a multiple of 8.
- The range must be a multiple of 8 bits.

F.5 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components

NOS/VE Ada does not support implementation-dependent names to be used in record representation clauses.

F.6 Interpretation of Expressions Appearing in Address Clauses

As permitted by the ANSI standard for Ada, NOS/VE Ada does not support address clauses and interrupts.

F.7 Restrictions on Unchecked Type Conversions

NOS/VE Ada allows unchecked conversions when objects of the source and target types have the same size.

F.8 Implementation-Dependent Characteristics of Input-Output Packages

The discussion of NOS/VE Ada implementation of input-output packages includes the following:

- External files and file objects
- NOS/VE Ada implemented exceptions for input-output errors
- Low level input-output

F.8.1 External Files and File Objects

NOS/VE Ada can process files created by another language processor as long as the data types and file structures are compatible.

NOS/VE Ada characterizes an external file as described in the ANSI Reference Manual for Ada, except as noted below:

- Name

  (see 14.2.1)

- Form

  Form strings have no effect on file processing in NOS/VE Ada. The FORM function returns the string you provided in the CREATE or OPEN procedure call.

NOS/VE Ada supports the following kinds of external files:

- Sequential access files (see 14.1)
- Direct access files (see 14.1)
- Text input-output files (see 14.3)

F.8.2 NOS/VE Ada Implemented Exceptions for Input-Output Errors

NOS/VE Ada raises the following language-defined exceptions for error conditions occurring during input-output operations:

- DATA_ERROR
- DEVICE_ERROR
- END_ERROR
- NAME_ERROR
- USE_ERROR

The DATA_ERROR is raised when:

- An attempt is made to read a direct file with a key that has not been defined

FIRST DRAFT INTERNAL USE ONLY

- A check reveals that the sizes of the records read from a file do not match the sizes of the Ada variables. NOS/VE Ada performs this check except in those few instances where it is too complicated to do so. NOS/VE Ada omits the check in these cases, as permitted by the ANSI Ada language definition. (see 14.2.2)

The DEVICE_ERROR is raised when:

- To be provided

The END_ERROR is raised when:

- An attempt is made to read a file beyond End_of_Information.

The NAME_ERROR is raised when:

- To be provided.

The USE_ERROR is raised when:

- The TEXT_IO package tries to process a file without the LIST attribute. (see 14.3)
- The NAME function tries to operate on a temporary file. (see 14.2.1)
- A file overflow condition exists.
- An attempt is made to delete an external direct file with multiple accesses while more than one instance of open is still active. The file remains open and the position is unchanged.

F.8.3 Low Level Input-Output

As permitted by the ANSI standard for Ada, NOS/VE Ada does not support the LOW_LEVEL_IO package.

F.9 Other Implementation-Dependent Characteristics

The other implementation-dependent characteristics of NOS/VE Ada are discussed as follows:

- Implementation features
- Entity types
- Tasking
- Interface to other languages
- Command interfaces
- Values of data attributes

F.9.1 Implementation Features

The NOS/VE Ada implementation features are listed as follows:

- Predefined types
- Basic types
- Compiler
- Definition of a main program
- TIME type
- Machine code insertions

F.9.1.1 Predefined Types

NOS/VE Ada implements all the predefined types required by the ANSI standard for Ada. It does not support the following predefined types:

- LONG_INTEGER
- SHORT_FLOAT
- SHORT_INTEGER

F.9.1.2 Basic Types

The sizes of the basic types are as follows:

Type	Size (bytes)
ENUMERATION	8
FIXED	8
FLOAT	8
INTEGER	8
LONG_FLOAT	16
TASK	8

The enumeration type encompasses booleans and characters as well as user defined enumerations. The boolean values FALSE and TRUE are defined as enumeration values 0 and 1, respectively.

F.9.1.3 Compiler

NOS/VE Ada provides an ANSI standard Ada compiler. The discussion of this compiler contains two types of information:

- The physical realization of the compiler
- Performance hints

F.9.1.3.1 Physical Realization of the Compiler

The NOS/VE Ada compiler supports the following:

- Up to 11 levels of nesting of concurrent tasks
- Source code lines up to 132 characters long
- Up to 100 static levels of nesting of blocks and/or subprograms
- External files up to one segment, 2**31-1 bytes, in length

F.9.1.3.2 Performance Hints

The compiler throughput improves when you submit multiple compilation units. However, if the number of compilation units grows over a certain limit, for example 50 small compilation units of about 50 lines each, or if the first compilation units are large, the throughput actually degrades.

Using the INLINE pragma, where applicable, results in faster object code by avoiding the call/return instructions and the dynamic initialization of the stack frame. For short subroutines, 10 to 15 source lines long, the size of the resulting object code might even be equivalent.

F.9.1.4 Definition of a Main Program

NOS/VE Ada requires that the main program be a procedure without parameters. The name of a compilation unit used as a main program must follow SCL naming standards. The name can be up to 31 characters in length and must be alphabetic. Any naming error is detected at link time only. For more details about naming the main program, see the Ada Usage manual.

F.9.1.5 TIME Type

The type TIME is defined as an integer representing the Julian date in milliseconds.

F.9.1.6 Machine Code Insertions

As permitted by the ANSI Ada language definition, NOS/VE Ada does not support machine code insertions.

F.9.2 Entity Types

This discussion contains information on:

- Array types
- Record types
- Access types

F.9.2.1 Array Types

Arrays are stored row wise, that is, the last index changes the fastest.

An array has a type descriptor that NOS/VE Ada uses when the array is one of the following:

- A component of a record with discriminants
- Passed as a parameter
- Created by an allocator

For each index, NOS/VE Ada makes the following triplet of information available:

```
----------------------------
|       LOWER BOUND        |
----------------------------
|       UPPER BOUND        |
----------------------------
|       ELEMENT SIZE       |
----------------------------
```

For multi-dimension arrays, NOS/VE Ada allocates the triplets consecutively.

Element size is expressed in number of storage units (64-bit words). If the array is packed, the element size is expressed in number of bits and represented by a negative value.

Strings are packed arrays of characters, and each element uses 8 bits (one byte). Packed arrays of booleans use 1 bit per element and are left justified. Arrays of integers or numeration variables can also be packed. Each element uses n bits, such that the integer or enumeration subtype is in the range -2**n .. 2**n-1.

Note that all objects start on a storage unit (64-bit word) boundary.

The maximum size of an array is limited by the amount of available space left either on the stack or in the heap at run-time when the object is allocated. (see 3.6)

F.9.2.2 Record Types

The maximum size of a record is limited by the amount of available space left either on the stack or in the heap at run-time when the object is allocated. (see 3.7)

The exception STORAGE_ERROR is raised at run time when the size of an elaborated object exceeds the amount of available space.

The rest of this discussion on how records are stored includes:

- Simple record types (without discriminants)
- Record types with discriminants

F.9.2.2.1 Simple Record Types (Without Discriminants)

In the absence of representation clauses, each record component is word aligned. NOS/VE Ada stores the record components in the order they are declared.

A fixed size array (lower and upper bounds are constants) is stored within the record. Otherwise, the array is stored elsewhere on the stack, and is replaced by a pointer to the array value (first element of the array) in the record.

F.9.2.2.2 Record Types With Discriminants

The discriminants are stored first, followed by all the other components as described for simple records.

If a record component is an array with index values that depend on the value of the discriminant(s), the array and its descriptor are both allocated on the heap. They are replaced by a pair of pointers in the record. One points to the array value and the other points to the array descriptor.

F.9.2.3 Access Types

Objects of access type are simply 6-byte pointers, left justified within a word, to the accessed data contained in some allocated area in the heap. If the accessed data is of type array or packed array, the allocated area also contains the address of the array descriptor in front of the data.

F.9.3 Tasking

NOS/VE Ada supports tasking by running all Ada tasks as NOS/VE concurrent procedures. The standard NOS/VE scheduling mechanism schedules the Ada tasks. See the Ada Usage manual for more description of NOS/VE Ada tasking.

F.9.4 Interface to Other Languages

NOS/VE Ada supports calls to CYBIL and FORTRAN subprograms and to NOS/VE MATH_LIBRARY subroutines with the following restrictions:

- CYBIL interface
  (see 13.9.1IDC and the Ada Usage manual)
- FORTRAN interface
  (see 13.9.2IDC and the Ada Usage manual)
- MATH_LIBRARY interface
  (see 13.9.3IDC and the Ada Usage manual)

F.9.5 Command Interfaces

The discussion of the command interfaces implemented by NOS/VE Ada includes

- Program Library Utility commands
- Compiler command
- Linker command

FIRST DRAFT INTERNAL USE ONLY

- Execution commands
- Dependency checker

F.9.5.1 Program Library Utility Commands

NOS/VE Ada provides a hierarchically structured Program Library to fulfill the ANSI Ada language definition requirements. The Ada Usage manual contains a detailed discussion of the NOS/VE Ada implementation of the Program Library.

F.9.5.2 Compiler Command

The NOS/VE Ada compiler can compile an ANSI standard Ada program on NOS/VE. See the Ada Usage manual to learn how to use the NOS/VE Ada compiler command.

F.9.5.3 Linker Command

NOS/VE Ada provides a linker to perform the following functions:

- Link an Ada program.
- Show the effects of a recompilation.

The Ada Usage manual tells more about the linker command and its use.

F.9.5.4 Execution Command

NOS/VE Ada provides several ways to load and execute an Ada program. They are described in the following manuals:

- Ada Usage
- CYBIL System Interface Usage
- SCL Object Code Management Usage

F.9.5.5 Dependency Checker

Ada compilation units can form a pattern of compilation dependencies where if you recompile some of the units, then you must recompile others. To make program changes correctly, you need to know this pattern.

NOS/VE Ada provides the LIST_ADA_DEPENDENCIES command to show you these compilation dependencies. The input to the command is a dependency list of compilation units. The output of the command is the list of the units you must recompile if you recompile the units in the input dependency list.

The Ada Usage manual tells how to use the LIST_ADA_DEPENDENCIES command

F.9.6 Values of Data Attributes

The package STANDARD contains the declaration of the following predefined types and their attributes:

- Integer (INTEGER)
- Floating point (FLOAT)
- Long floating point (LONG_FLOAT)
- Fixed point (FIXED)
- Duration (DURATION)

F.9.6.1 Values of Integer Attributes

Attribute	Value
FIRST	-9_223_372_036_854_775_808
LAST	9_223_372_036_854_775_807
SIZE	64

F.9.6.2 Values of Floating Point Attributes

Attribute	Value
DIGITS	13
FIRST	-16#7.FFFF_FFFF_FFF8#E1023
LAST	16#7.FFFF_FFFF_FFF8#E1023
MACHINE_EMAX	4095
MACHINE_EMIN	-4096
MACHINE_MANTISSA	48
MACHINE_OVERFLOWS	TRUE
MACHINE_RADIX	2
MACHINE_ROUNDS	FALSE
SAFE_EMAX	4095
SAFE_LARGE	16#7.FFFF_FFFF_FFC#E1023
SAFE_SMALL	16#1.0#E-1024
SIZE	64

F.9.6.3 Values of Long Floating Point Attributes

Attribute	Value
DIGITS	28

FIRST	-16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023
LAST	16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023
MACHINE_EMAX	4095
MACHINE_EMIN	-4096
MACHINE_MANTISSA	96
MACHINE_OVERFLOWS	TRUE
MACHINE_RADIX	2
MACHINE_ROUNDS	FALSE
SAFE_EMAX	4095
SAFE_LARGE	16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF#E1023
SAFE_SMALL	16#1.0#E-1024
SIZE	128

F.9.6.4 Values of Fixed Point Attributes

Attribute	Value
FIRST	-9_223_372_036_854_775_808 (SMALL=1)
LAST	9_223_372_036_854_775_807 (SMALL=1)
SIZE	64
MACHINE_ROUNDS	FALSE
MACHINE_OVERFLOWS	TRUE

F.9.6.5 Values of Duration Attributes

Attribute	Value
FIRST	-6_279_897_600.0
LAST	6_279_897_600.0
DELTA	0.001
SMALL	2#1.0#E-10
SMALL_POWER	-10

FIRST DRAFT INTERNAL USE ONLY

3.0 ADA COMPILER FEATURES

The Ada compiler features are governed by the DoD ANSI/MIL-STD-1815A standard, and are developed accordingly.

The following list of restrictions and implementation dependencies represents the material for Appendix F of the CYBER 180 Ada Usage Manual. Appendix F describes all the implementation dependent characteristics of a given Ada compiler.

3.1 MACHINE DEPENDENT CHARACTERISTICS

section 2.2 - the length of a line is limited to 132 characters.

section 3.5.4 - the predefined types SHORT_INTEGER and LONG_INTEGER are not supported.

section 3.5.7 - the predefined type SHORT_FLOAT is not supported.

section 3.6 & 3.7 - the maximum size of an array or a record will be limited by the amount of available space left either on the stack or in the heap at run-time when the object is allocated. Stack and heap are allocated on a segment each. Although the maximum length of a segment is 2**31-1 (2_147_483_647) bytes, the default for the size of the stack will generally have been set to a lower value by the installation and the user may have to increase the maximum size of the stack by setting the program attribute PMCSMAX_SIZE_SPECIFIED in order to be able to execute large Ada application programs.

3.2 IMPLEMENTATION DEPENDENT FEATURES

section 3.7.2 - the exception STORAGE_ERROR is raised at run time when the size of an elaborated object exceeds the amount of available space.

section 9.6 - the type TIME is defined as an integer representing the Julian date in milliseconds.

3.0 ADA COMPILER FEATURES
3.2 IMPLEMENTATION DEPENDENT FEATURES

section 10.1 - the name of a compilation unit that is used as a main program is limited to 31 characters. The error is detected at link time only.

section 12.1 - generic declarations

a generic library package body cannot be compiled in a different file from its specification.

a generic non-library package body or a generic non-library subprogram body cannot be compiled as a subunit in a separate file from its specification.

section 13.1 - representation clauses
no representation clauses may be given for a derived type.

section 13.2 - length clauses

SIZE: not supported

STORAGE SIZE (collection size): supported

STORAGE SIZE (task): supported

SMALL: not supported. The compiler always chooses for SMALL the largest power of 2 not greater than the DELTA of the fixed_accuracy_definition.

section 13.3 - Enumeration representation clauses
internal codes must be in the range of the predefined type INTEGER.

section 13.4 - Record representation clauses

ALIGNMENT_CLAUSE : not supported.

COMPONENT_CLAUSE :
at: giving the address of the starting storage unit of a component of a record, relative to the beginning of the record.
range: giving the bit positions occupied by the record component in the storage unit defined by the "at" component_clause, is supported for discrete types only (integers or enumerations).

At Release 1, the 'range' must be 16 bits or less. If the 'range'of a record component is greater than or equal to 8 bits, then the 'range' must start on a multiple of 8 bits.

3.0 ADA COMPILER FEATURES
3.2 IMPLEMENTATION DEPENDENT FEATURES

---

section 13.5 - address clause: an address in storage cannot be specified for any entity and interrupts are not supported.

section 13.7 - the specifications of the predefined library package SYSTEM are:

```
package SYSTEM is
    type ADDRESS is access INTEGER;
    type NAME is (CYBER180);

    SYSTEM_NAME : constant NAME:= CYBER180;

    STORAGE_UNIT : constant:=64;
    MEMORY_SIZE : constant:=128*1048576;-- 128 megabytes

    MIN_INT   :   constant:=   -9_223_372_036_854_775_808;
    --(-2**63)
    MAX_INT   :   constant:=    9_223_372_036_854_775_807;
    --2**63-1
    MAX_DIGITS : constant:= 28;
    MAX_MANTISSA : constant:= 63;
    FINE_DELTA : constant:= 2#1.0#E-63; --2**(-63)
    TICK : constant:=0.001;

    subtype PRIORITY is INTEGER range 0..0;
    note: NOS/VE support for changeable task priority may
    become available on time for Release 1 but is not
    required for Validation.

end SYSTEM;
```

section 13.8 - machine code insertions are not supported.

section 13.9 - interface to other languages

calls to CYBIL and FORTRAN subprograms and to CYBER 180 MATH_LIBRARY subroutines are supported at Release 1 with the following restrictions:

CYBIL interface

CYBIL allows either value or variable parameters for both procedures and functions.

The Ada compiler passes scalar parameters by value and array and string parameters by reference. When calling a CYBIL subprogram, the Ada compiler passes pointers to a copy of the values of the scalar parameters and pointers to the actual values of the

array and string parameters, i.e. Ada supports only CYBIL calls by reference.

The Ada rules for scalar parameters are obeyed since the CYBIL code has only access to copies. However, arrays and strings passed as actual parameters to a CYBIL subprogram are not protected.

All Ada parameter modes are allowed on a call to a CYBIL subprogram and there is no cross-type checking between the Ada call actual parameters and the CYBIL subprogram formal parameters.

In Ada, the size of an array or string parameter is always known at the time of a call. CYBIL expects an address (PVA) for fixed size arrays and strings greater than one word, a value for fixed size arrays and strings less than one word in length and passed as value parameters, an adaptable pointer for adaptable arrays and strings passed as parameters. The Ada compiler does not know the characteristics of the corresponding CYBIL parameters and consequently Ada supports only adaptable arrays and strings type CYBIL parameters.

3.0 ADA COMPILER FEATURES
3.2 IMPLEMENTATION DEPENDENT FEATURES

The data types correspondances between Ada and CYBIL are given in the following table:

Ada	CYBIL
Integer	Integer
Float	Real
Long_Float	Long_Real (1.3.1)
Access	Pointer
Integer	Integer
Float	Real
Long_float	Long_Real
Address	Pointer to cell
Enumeration (<256)	Ordinal
Boolean	Boolean
Character	Char
String	Adaptable String
Array	Adaptable Array
Integer	Integer
Float	Real
Long_float	Long_Real

FORTRAN interface

FORTRAN subroutines and functions expect parameters to be passed by reference.

The Ada compiler passes scalar parameters by value and array and string parameters by reference.

When calling a FORTRAN subprogram, the Ada compiler passes pointers to a copy of the value for scalar parameters and pointers to the actual values for the other types of parameters.

The modes and types of the Ada actual parameters and FORTRAN formal parameters are not cross-checked.

Since in Ada the length of an array or a string parameter is always known at the time of the subprogram call, the Ada compiler can pass the length of a string with the address of the string or the array descriptor with the address of the array. This allows parameters of type array or string to be declared in FORTRAN as either fixed or conforming size.

3.0 ADA COMPILER FEATURES
3.2 IMPLEMENTATION DEPENDENT FEATURES

The data type correspondences between Ada and FORTRAN are given in the following table:

Ada	FORTRAN
Integer	Integer, Boolean
Float	Real, Boolean
Long_Float	Double
Boolean	Logical
String	Character (fixed or conforming)
Array	Array (fixed or conforming
Integer	Integer, Boolean
Float	Real, Boolean
Long_Float	Double

MATH_LIBRARY interface

the call by reference interface is supported.

section 13.10.2 - unchecked conversions are allowed, at Release 1, when objects of the source and target types have the same size.

3.3 INPUT-OUTPUT

section 14.1 - External files and file objects

Ada can process files created by another language processor as long as the data types and file structures are compatible.

Text Input_Output

A text file associated with a terminal device can be shared. A text file associated with an external file cannot be shared.

Terminal files:

input: all the data entered on the terminal is transmitted to the user.

output: information (format effectors, CR) is added in front of the output for the terminal usage (column 1). All the user's output data is displayed at the terminal.

Non-terminal files:

TEXT_IO processes only files with the LIST attribute. A predefined exception USE_ERROR is raised for non-list files. A file created by the NOS/VE editor, i.e. LEGIBLE, cannot be read by TEXT_IO. It must first be copied to a file with the LIST attribute.

Data in column 1 is interpreted as format effector or CR on input, and conversely a format effector or CR is added in front of the output data (column 1).

Text files can be dynamically created and deleted.

Sequential access

External files cannot be shared.

SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants and no default values for the discriminants.

Constrained records 64,000 bytes or less and other fixed size objects are written as F record types. Constrained records larger than 64,000 bytes or unconstrained records are written as V record types.

Direct access

External files can be shared. However, only one of the files associated with an external file can specify mode INOUT_FILE or OUT_FILE.

DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants and no default values for the discriminants.

Direct files are implemented as Indexed-Sequential files. They use nonembedded, uncollated, 8-byte keys divided into:

record index: 4 bytes

parcel count: 2 bytes
flags: 1 byte
unused bit count: 1 byte

The record index is the index specified in the Ada WRITE procedure call.

Records larger than 64,000 bytes are broken in parcels of 64,000 or less bytes. The parcel count is one for records less than 64,000 bytes long.

Flags are used to mark the last parcel of a multi-parcel record.

The unused bit count is used for records that are fields less than 64 bits long.

section 14.2.1 - file management

NAME can be :

.a string referencing an external file, i.e. a NOS/VE file, that will be associated with the Ada file. NAME must be a valid NOS/VE file specification conform to the NOS/VE file naming conventions. The external file will exist after program termination and may be accessed later from another Ada program.

.a null string specifying a temporary file that will not be accessible after job termination. A NAME function on a temporary file will raise a USE_ERROR exception.

FORM: has no action at Release 1. The FORM function will return the string provided by the user in the CREATE or OPEN procedure call.

An attempt to read a direct file with a key that has not been defined raises DATA_ERROR.

A file overflow condition raises USE_ERROR.

An attempt to read a file beyond End_Of_Information raises END_ERROR.

An attempt to create a sequential or direct file with mode IN_FILE raises USE_ERROR.

An attempt to delete an external direct file with multiple accesses raises USE_ERROR if more than one instance of

open is still active. The file remains open and the position is unchanged.

In most cases, the run-time input-output packages check that the sizes of the records read from a file and the Ada variables match. There are a few cases where performing the check would be too complex and the CYBER 180 implementation omits the check as allowed by the Ada language definition.

If the sizes do not match, DATA_ERROR is raised. However the data is always transfered except for scalars read from a text file.

section 14.6 - Low Level Input-Output
The package LOW_LEVEL_IO is not implemented on CYBER 180.

3.4 APPENDIXES

appendix A. Predefined Language Attributes

P'ADDRESS - The 48-bit PVA for the prefix P is returned, right justified within a 64-bit INTEGER.

appendix B. Predefined Language Pragmas

CONTROLLED - This pragma has no effect.

ELABORATE - This pragma is supported as described in the Ada Reference Manual.

INLINE - This pragma causes inline expansion of a subroutine except in the following cases:

1. The body of the subroutine to be expanded inline has not yet been entirely seen. This protects against the inline expansion of recursive subprograms.

2. The subprogram call appears in an expression on which conformance check may be applied (e.g. subprogram specification, discriminant part, formal part of an entry declaration or accept statement).

3. The subprogram is an instantiation of the predefined generic subprograms UNCHECKED_CONVERSION or UNCHECKED_DEALLOCATION.

4. The subprogram is declared in a generic unit. The body of that generic unit is compiled as a secondary unit in the same compilation as the unit containing a call to (an instance of) the subprogram.

5. The subprogram is declared by a renaming declaration.

6. The subprogram is passed as a generic actual parameter.

A warning is given if inline expansion is not performed.

INTERFACE - This pragma is supported for FORTRAN, CYBIL, and the CYBER 180 MATH_LIBRARY.

LIST - This pragma is supported as described in the Ada Reference Manual.

MEMORY_SIZE - This pragma is not supported.

OPTIMIZE - This pragma has no effect.

PACK - Objects of the given type are packed into the nearest 2**n bits.

PAGE - This pragma is supported as described in the Ada Reference Manual.

PRIORITY - This pragma has no effect.

SHARED - This pragma is not allowed for variables of type LONG_FLOAT or subtypes or derived types of thereof.

SUPPRESS - This pragma is supported but it is not possible to restrict the omission of a check to a specific object or type.

STORAGE_UNIT - This pragma is not supported.

SYSTEM_NAME - This pragma is not supported.

appendix C. Predefined Language Environment

Package STANDARD contains the declaration of the following predefined types and their attributes:

3.0 ADA COMPILER FEATURES
3.4 APPENDIXES

```
type INTEGER
    INTEGER'FIRST = -9_223_372_036_854_775_808
    INTEGER'LAST = 9_223_372_036_854_775_807
    INTEGER'SIZE = 64

type FLOAT
    FLOAT'DIGITS = 13
    FLOAT'FIRST = -16#7.FFFF_FFFF_FFF8#E1023
    FLOAT'LAST = 16#7.FFFF_FFFF_FFF8#E1023
    FLOAT'MACHINE_EMAX = 4095
    FLOAT'MACHINE_EMIN = -4096
    FLOAT'MACHINE_MANTISSA= 48
    FLOAT'MACHINE_OVERFLOWS= TRUE
    FLOAT'MACHINE_RADIX = 2
    FLOAT'MACHINE_ROUNDS = FALSE
    FLOAT'SAFE_EMAX = 4095
    FLOAT'SAFE_LARGE = 16#7.FFFF_FFFF_FFC#E1023
    FLOAT'SAFE_SMALL = 16#1.0#E-1024
    FLOAT'SIZE = 64

type LONG_FLOAT
    LONG_FLOAT'DIGITS = 28
    LONG_FLOAT'FIRST =
              -16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023
    LONG_FLOAT'LAST =
              16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF8#E1023
    LONG_FLOAT'MACHINE_EMAX = 4095
    LONG_FLOAT'MACHINE_EMIN = -4096
    LONG_FLOAT'MACHINE_MANTISSA = 96
    LONG_FLOAT'MACHINE_OVERFLOWS = TRUE
    LONG_FLOAT'MACHINE_RADIX = 2
    LONG_FLOAT'MACHINE_ROUNDS = FALSE
    LONG_FLOAT'SAFE_EMAX = 4095
    LONG_FLOAT'SAFE_LARGE =
              16#7.FFFF_FFFF_FFFF_FFFF_FFFF_FFF#E1023
    LONG_FLOAT'SAFE_SMALL = 16#1.0#E-1024
    LONG_FLOAT'SIZE = 128

type FIXED
    FIXED'FIRST = -9_223_372_036_854_775_808 (SMALL=1)
    FIXED'LAST = 9_223_372_036_854_775_807 (SMALL=1)
    FIXED'SIZE = 64
    FIXED'MACHINE_ROUNDS = FALSE
    FIXED'MACHINE_OVERFLOWS = TRUE

type DURATION
    DURATION'FIRST = -6_279_897_600.0
    DURATION'LAST = 6_279_897_600.0
    DURATION'DELTA = 0.001
```

```
DURATION'SMALL = 2#1.0#E-10
DURATION'SMALL_POWER = -10
DURATION'SIZE = 64
DURATION'MACHINE_ROUNDS = FALSE
DURATION'MACHINE_OVERFLOWS = TRUE
```

5.0 INTERFACES

5.1 COMMAND INTERFACES

5.1.1 COMPILER COMMAND

This command uses the syntax and language elements for parameters described in the SCL Language Definition manual.

Purpose

The ADA command invokes the compiler, specifies the current sublibrary, the files to be used, and the compiler options that have been selected.

Format

```
ADA
   INPUT= file reference
   PROGRAM_LIBRARY= file reference
   LIST= file reference
   DEBUG_AIDS= list of keyword values
   ERROR= file reference
   ERROR_LEVEL= keyword value
   LIST_OPTIONS= keyword value
   OPTIMIZATION_LEVEL= keyword value
   STATUS= status variable
```

Parameters

INPUT (I)
specifies the file that contains the source text to be read. The source input ends when an end_of_partition or an end_of_information is encountered on the source input file. The default value is $INPUT.

LIST (L)
specifies the file where the compiler writes the source listing, diagnostics, statistics, and any additional list information specified by the LO parameter. The default value is $LIST.

PROGRAM_LIBRARY (PL)
a file name assigned by the user to the current sublibrary at creation time. The parameter cannot be omitted.

DEBUG_AIDS (DA)
specifies the debug options to be selected.

5.0 INTERFACES
5.1.1 COMPILER COMMAND

ALL All of the available options are selected for the DEBUG parameter.

DS Compile all debugging statements. Debugging statements will be implementation defined Pragmas specifying the compile time directives for the Debugger.

DT Generate line numbers and symbol table as part of the object code.

NONE No debug tables are produced.

If the parameter is omitted, NONE is assumed.

ERROR (E)
specifies the file to receive the error listing. The default value is $ERRORS.

ERROR_LEVEL (EL)
Indicates the minimum severity level of the diagnostics to be listed. The levels, in increasing order of severity, are:

W Warning - An error that does not change the meaning of the program or hinder the generation of object code. Also a construct for which the object code will raise a CONSTRAINT_ERROR at run time.

F Fatal - An illegal construct in the source program has been detected. The compilation continues but no object code will be generated.

C Catastrophic - An error that causes the compiler to be terminated immediately. No object code will be generated.

If the parameter is omitted, W is assumed.

LIST_OPTIONS (LO)
specifies what information will appear on the listing file (LIST parameter). Multiple options may be specified.

O Object code listing

R Symbolic cross-reference listing of all program

entities

S Source input listing

NONE No list options are selected.

If the parameter is omitted, S is assumed.

OPTIMIZATION_LEVEL (OL)
specifies the level of object code optimization.

DEBUG Object code is stylized to facilitate debugging.

LOW Lowest level of production quality code with only AA-code stack optimization.

HIGH High level of production quality code obtained thru optimization of the AA-code, detection of common subexpressions, elimination of unnecessary range checking.

OL=LOW, the default option, is the only optimization option supported at Release 1.

STATUS
specifies the name of the SCL status variable to be set by the compiler at completion time.

5.1.2 LINKER COMMAND

This command uses the syntax and language elements for parameters described in the SCL Language Definition manual.

Purpose

The LINK_ADA command calls the linker, specifies the files to be used, and identifies the subprogram, which must be a library unit, that will become the main program.

The linker collects from the program library the binaries of the main program and of all the library units that have been named by the main program, directly or indirectly, in context clauses.

The linker verifies that the compilation order rules have been obeyed by comparing the time stamps recorded

for each library unit in the program library.

The linker also generates a sequence of code that will, at execution time, initialize the Ada Run-Time System, elaborate the library units in the proper order, and call the main program.

Format

```
LINK_ADA or LINA
   MAIN_PROGRAM= name
   PROGRAM_LIBRARY= file reference
   BINARY= file reference
   STATUS= status variable
```

Parameters

MAIN_PROGRAM (MP)
specifies the name, as defined in the Ada source code, of the subprogram to be executed. The subprogram must have been previously compiled as a library unit into the sublibrary specified by the PROGRAM_LIBRARY parameter. The parameter cannot be omitted.
Note: at Release 1, the main program must be a parameterless procedure.

PROGRAM_LIBRARY (PL)
specifies the file containing the user's current sublibrary. The parameter cannot be omitted.

BINARY (B)
specifies the file on which the binaries extracted from the user's program library will be written. If $NULL is specified, the Ada linker will perform all the compilation order validation checks, but will not create a binary file. The default value is $LOCAL.LGO.

STATUS
specifies the name of the SCL status variable to be set by the linker at completion time.

5.1.3 EXECUTION COMMAND

An Ada program can be loaded and executed in the following ways:

.a direct file reference resulting in the binary file

5.0 INTERFACES
5.1.3 EXECUTION COMMAND

---

being loaded and executed, e.g.;
BINARY
BINARY being the name assigned to the Linker output file.

.an SCL command:
EXECUTE_TASK
FILES=file reference

FILES specifies the object file generated by the Linker.

.a call to the system interface procedure PMP$EXECUTE:
The object file generated by the Linker will be referenced in the object_file_list variable of the program_description parameter.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	<131 x "A">1
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	<131 x "A">2
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	<65 x "A">3<66 x"A">
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	<65 x "A">4<66 x"A">
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	<129 x "0">298
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	<126 x "0">69.0E1

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	<66 x "A">
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	<65 x "A">1
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	<112 x " ">
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	9223372036854775807
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	67
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD_CHARS^#.%!X
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	This_File_Name_Has_To_Be_Too_ Long_Wild_Card_Char_Do_Not_Exist
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	7000000000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHARS^@.~!

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	MUCH_TOO_LONG_NAME_FOR_A_VE_ FILE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-9_223_372_036_854_775_808
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	9_223_372_036_854_775_807
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	9223372036854775808
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	100000000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-7000000000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	28
$MAX_IN_LEN Maximum input line length permitted by the implementation.	132
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	9223372036854775807
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	9223372036854775808
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	2:<127 X "0">11:

$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	16:<125 X "0">F.E:
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	"<130 X "A">"
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-9223372036854775808
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	DOES_NOT_EXIST
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	8#1<20 x "7">6

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 25 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A A basic declaration (line 36) wrongly follows a later declaration.

E28005C This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ALMP.

C34004A The expression in line 168 wrongly yield a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P The equality operators in lines 62 and 69 should be inequality operators

A35902C Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35A03E & R These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

C37213H The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E,G,H Various discriminant constraints are wrongly expected to be incompatible with the type CONS.

C38102C The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A	'STORAGE_SIZE is wrongly applied to an object of an access type.
C45614C	REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).
A74106C C85018B C87B04B CC1311B	A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur about lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere)
BC3105A	Lines 159..168 are wrongly expected to be incorrect; they are correct.
AD1A01A	The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.
CE2401H	The record aggregates in lines 105 and 117 contain the wrong values.
CE3208A	This test expects that an attempt to open the default output file (after it was closed) with MODE_IN file raises NAME_ERROR or USE_ERROR; by commentary AI-00048, MODE_ERROR should be raised.